09772466 . 012601

MAP Decoding with Parallelized Sliding Window Processing

U.S. Patent Application of:

Alan Gatherer, Inventor
Tod D. Wolf, Inventor

Texas Instruments Incorporated, Assignee

Attorney's Docket No. TI-30024

MAP Decoding with Parallelized Sliding Window Processing

Background and Summary of the Invention

The present application relates to wireless communication, and more particularly to turbo decoding and the like.

Background: Error Correction

Coded digital communication systems use error control codes to improve data reliability at a given signal-to-noise ratio (SNR). For example, an extremely simple form (used in data storage applications) is to generate and transmit a parity bit with every eight bits of data; by checking parity on each block of nine bits, single-bit errors can be detected. (By adding three error-correction bits to each block, single-bit errors can be detected and corrected.) In general, error control coding includes a large variety of techniques for generating extra bits to accompany a data stream, allowing errors in the data stream to be detected and possibly corrected.

Background: Trellis Coding

One of the important techniques for error control is trellis coding. In this class of techniques some constraints are imposed on the sequence of symbols, so that certain symbols cannot be directly followed by others. The constraints are often defined by a geometrical pattern (or "trellis") of allowed and disallowed transitions. The existence of constraints on the sequence of symbols provides some structure to the data sequence: by analyzing whether the constraints have been violated, multiple errors can be corrected. This is a very powerful class of coding techniques; the constraint geometry can be higher dimensional, or algebraic formulations can be used to express the constraints, and many variations can be used.

Background: Turbo Coding

The encoder side of a turbo coding architecture typically uses two encoders, one operating on the raw data stream and one on a shuffled copy of the base data stream, to generate two parity bits for each bit of the raw data stream. The encoder output thus contains three times as many bits as the incoming data stream. This "parallel concatenated encoder" (or "PCE") configuration is described in detail below.

The most surprising part of turbo coding was its decoding architecture. The decoder side invokes a process which (if the channel were noiseless) would merely reverse the transformation performed on the encoder side, to reproduce the original data. However, the decoder side is configured to operate on soft estimates of the information bits and refines the estimates through an iterative reestimation process. The decoder does not have to reach a decision on its first pass, but is generally allowed to iteratively improve the estimates of the information bits until convergence is achieved.

Background: MAP Decoders

MAP decoding is a computationally intensive technique, which has turned out to be very important for turbo decoding and for trellis-coded modulation. "MAP" stands for "maximum a posteriori": a MAP decoder outputs the most likely estimate for each symbol in view of earlier AND LATER received symbols. This is particularly important where trellis coding is used, since the estimate for each symbol is related to the estimates for following symbols.

By contrast, a maximum-likelihood ("ML") decoder tries to compute the transmitted sequence for which the actually received sequence was most likely. These verbal statements may sound similar, but the difference between MAP and ML decoding is very significant. ML decoding is computationally simpler, but in many applications MAP decoding is required.

MAP decoding normally combines forward- and back-propagated estimates: a sequence of received symbols is stored, and then processed in one direction (e.g. forward in time) to produce a sequence of forward transition probabilities, and then processed in the opposite direction (backward in time) to produce a sequence of backward transition probabilities. The net estimate for each symbol is generated by combining the forward and backward transition probabilities with the data for the signal actually received. (Further details of this procedure can be found in OPTIMAL DECODING OF LINEAR CODES FOR MINIMIZING SYMBOL ERROR RATE, Bahl, Cocke, Jelinek, and Raviv, *IEEE Transactions on Information Theory*, 1974, which is hereby incorporated by reference.)

The combination of forward and backward computation requires a substantial amount of memory. Since the blocks in advanced cellular communications can be large (e.g. 5120 symbols), the memory required to store a value for each possible transition for each symbol in a block is large. To reduce the memory requirements during decoding, each block of data may be divided into many smaller blocks (e.g. 40 blocks of 128 symbols) for MAP decoding.

The trellis encoding is done on a complete block of data, so that starting and ending states are known for the complete block. However, the starting and ending states are not know for the intermediate blocks. This presents a problem for accurate process of these smaller blocks, but it has been found that simply iterating the forward estimation process for a few symbols before the start of each block will ensure that processing of the first symbol in the block starts from a good set of initial values.

MAP Decoding with Pipelined Windowed Processing

The present application discloses a technique for sub-block processing, in a MAP decoding, which uses pipelining. Processing of alphas is begun, in parallel with processing of betas. Preferably each stage of processing is further internally parallelized; but the pipelining of forward-propagated processing with back-propagated processing provides an additional degree of net improvement in throughput.

Advantages of the disclosed methods and structures, in various embodiments, can include one or more of the following:

- Faster processing;
- less memory;
- more iterations possible in a turbo decoder.

Brief Description of the Drawings

The disclosed inventions will be described with reference to the accompanying drawings, which show important sample embodiments of the invention and which are incorporated in the specification hereof by reference, wherein:

Figure 1 shows a block diagram of a turbo decoder.

Figure 2 shows a block diagram of a MAP decoder that uses parallel sliding window processing.

Figure 3 shows a block diagram of the beta generation block within the MAP decoder.

Figure 4 shows a block diagram of the alpha generation block within the MAP decoder.

Figure 5 shows a block diagram of the extrinsic generation block within the MAP decoder.

Figure 6 is a timing chart of the pipelining within the beta block.

Figure 7 shows the timing offset between generation of alpha and beta sliding window blocks.

Figure 8 shows the correspondence between the alpha and beta sliding window blocks, with prologs.

Figure 9 shows an example of the order in which beta and alpha bits are processed.

Detailed Description of the Preferred Embodiments

The numerous innovative teachings of the present application will be described with particular reference to the presently preferred embodiment. However, it should be understood that this class of embodiments provides only a few examples of the many advantageous uses of the innovative teachings herein. In general, statements made in the specification of the present application do not necessarily delimit any of the various claimed inventions. Moreover, some statements may apply to some inventive features but not to others.

Concurrent operation of system hardware allows simultaneous processing of more than one basic operation. Concurrent processing is often implemented with two well known techniques: parallelism and pipelining.

Parallelism includes replicating a hardware structure in a system. Performance is improved by having multiple structures execute simultaneously on different parts of a problem to be solved.

Pipelining splits the function to be performed into smaller pieces and allocates separate hardware to each piece. More information on parallelism and pipelining can be found in the "The Architecture of Pipelined Computers," by Kogge, which is hereby incorporated by reference.

Figure 1 shows a block diagram of a turbo decoder. Two main blocks, the turbo controller **102** and the MAP decoder **104**, are shown.

The turbo controller 102 stores the data streams (X, the systematic data **106**; P, the parity data **108**; and A, the A PRIORI data **110**) that serve as input for the MAP decoder 104 and controls the order in which the data is input in the MAP decoder 104. The diagram shows the three data streams being input twice each in the MAP decoder 104. Two separate sets of input data are required because the alpha and beta generation blocks require the data inputs in reverse order. The extrinsic output of the MAP decoder 104 is returned to the controller 102 for another decoding iteration.

Figure 2 shows a block diagram of a MAP decoder 104 that uses parallel sliding window processing. A MAP decoder 104 receives the scaled systematic data signal 106, the scaled parity data signal 108, and the A PRIORI signal 110 as its input. There are N number of X signals 106, where N is the size of the interleaver. N X signals 106 are applied for each beta **208** and alpha **210** state vector, which are the respective outputs of the beta **202** and alpha **206** blocks. During beta generation, X 106 is applied in reverse order, and during alpha generation, X 106 is applied in forward order. There are also N number of P signals 108. N P signals 108 are applied for each alpha 210 and beta 208 vector. During beta generation, P 108 is applied in reverse order, and during alpha generation it is applied in forward order. The A PRIORI 110 is either the interleaved or deinterleaved extrinsic data from the previous MAP decoder operation. There are N A PRIORI signals 110, and one A PRIORI 110 is applied for each beta 208 and alpha 210 vector. A PRIORI 110 is applied the same directions as X 106 and P 108 for beta and alpha generation.

The beta generation section 202, shown in more detail in **Figure 3**, receives inputs X 106, P 108, and A 110. It generates the beta state vector 208, which is stored in beta RAM **204**. The alpha generation section 206 receives inputs X 106, P 108, and A 110 (but in reverse order relative to the beta input). The alpha generation block 206, shown in greater detail in **Figure 4**, generates the alpha state vector 210. The outputs 208, 210 of the alpha and beta generation sections serve as inputs for the extrinsic generation section **212**, shown in **Figure 5**. These data streams must be properly sequenced with the parity stream P 108 before being input to the extrinsic section 212.

Figure 3 shows the beta generation stages. First, during the MAP reset state, the registers are set to their initial conditions for the beta state vector 208. The beta signal 208, X 106, P 108, and A 110 are summed together by the adder tree **302** according to the trellis used to encode the data. (In the preferred embodiment, an 8 state trellis is used). The results are stored in registers **310**. In the second stage, the results of the adder 302 are applied to the 8 MAX* blocks **304**, and are then stored in the MAX* registers 312. Next, the unnormalized outputs advance to two separate normalization stages **306, 308**, each of which has a register **314, 316** to store its results. Thus the total process has 4 stages within the feedback loop of the beta generation block 202, which require 4 clock cycles to complete. This latency (the 4 clock cycles) determines the level of pipelining available.

The alpha generation section 206 is shown in **Figure 4**. First, the registers are set to their initial conditions. Then the data inputs are summed together by the adder **402**, and the results are stored in registers **412**. These are then input to the MAX* blocks **406** and stored in MAX* registers **414**. The alpha generation section 206 also has two normalization stages **408, 410**, each with their own registers **416, 418**. The latency of the alpha generation stage 206 is thus 4, allowing 4 levels of pipelining to be implemented.

Operating in parallel with the alpha generation section is the extrinsic generation section 212, shown in **Figure 5**. Alpha 210, beta 208, and P 108 are summed together by the adders **502** according to the trellis used, and the results are stored in registers **510**. In the second stage, these results are applied to the MAX* blocks **504**, and then stored in the MAX* registers **512**. These results are again applied to MAX* blocks **504** and then stored in registers **508**. The result is summed and stored in another register **514**, and the output is the extrinsic signal **214**.

Parallelism with Sliding Windows

A sliding window approach basically consists of dividing the N sized block of incoming data into several smaller blocks. Each of these smaller blocks is called a sliding window block. These sliding window blocks are MAP decoded each independently, with a prolog for both the alpha and beta vectors. The decoding for the individual alpha and beta sliding window blocks is done in parallel. Since the initial conditions are not known for the individual sliding window blocks, the prologs are used to reach a good set of initial values.

By starting the update of the alpha at a point sufficiently inside the previous block and starting the update of the beta at a point sufficiently inside the next block, the decoder can "forget" the initial conditions and converge before it begins operating on the actual data. the prolog section size used is generally 3 or 4 times the number of states in the trellis. The first alpha and last beta sliding block will originate from a known state, and the size of their respective prolog sections will be 3 for an 8 state trellis (for example).

The innovative alpha prolog allows parallel processing of both the alpha and beta sliding window blocks of data. Depending on the specific implementation used, each update of alpha or beta takes a few clock cycles to run (4 clock cycles in the above embodiment). This latency determines the degree of pipelining possible in the system. In the preferred embodiment, there are four levels of pipelining within each alpha and beta block (meaning the data within each of the alpha and beta generation stages is pipelined, or broken into separate sets of data and independently operated on by successive stages within the beta generation section). There is also a degree of parallelism between the alpha and beta blocks themselves, meaning these two sections operate simultaneously to produce extrinsic input.

The alpha and beta vector generation processes are divided into multiple stages, as shown above. These stages are within the iteration loops of the alpha and beta vector generation, shown in **Figures 3** and **4**. The number of stages would be equal to the latency for a particular architecture. In the preferred embodiment, these stages are the Adder, the MAX*, and two Normalization stages. The latency of these stages dictates the degree of parallel processing possible. For example, in the preferred embodiment this latency is 4, meaning 4 sliding-window blocks can be processed in parallel. Thus, 4 sliding-window blocks make up one sub-block.

The pipelining of the sliding blocks is shown in **Figure 6**. During the first clock cycle, beta0 (the first sliding block) enters the adder stage. In the second clock cycle, beta0 enters the MAX* stage, and beta1 enters the adder stage. In the third clock cycle, beta0 enters the first normalization stage (the third stage of beta generation), beta1 enters the MAX* stage, and beta2 enters the adder stage. Next, beta0 enters the second normalization stage, beta1 enters the first normalization stage, beta2 enters the MAX* stage, and beta3 enters the adder stage. The intermediate values for each stage are stored in registers, as shown above.

Either the beta or alpha stages are stored in memory so that the data input to the extrinsic section can be synchronized. In the preferred embodiment, beta processing begins one sub-block before alpha processing (Note that this staggering could be eliminated by adding another RAM block to store the alpha outputs.) This staggering is shown in **Figure 7**. The first sub-block (which is a number of sliding blocks equal to the latency of the architecture--4 in the preferred embodiment) of the beta section can be processed while the alpha section is idle. Next, the second set of sliding blocks of the beta section (i.e., the second sub-block) is processed while the first set of sliding blocks of the alpha section are processed. The extrinsic sections are processed in parallel with the alpha section. This reduces the memory requirement for storing both the alpha and beta state vectors because the alpha outputs can be directly applied to the extrinsic as they are generated. Since the extrinsic generates output (and requires input) one sub-block at a time, the beta RAM only needs to store one sub-block of data at a time. (Note that the alpha and beta processing could be reversed. This would require the alpha outputs to be stored in memory, and beta and the extrinsic blocks would run in parallel.)

Figure 8 shows the correspondence between alpha and beta sliding-window blocks. The entire data block consists of N symbols plus a number of tail bits. This block is broken into sub-blocks, which are further divided into sliding-window blocks. One sliding-window block is processed per clock cycle. Each sliding-window block includes a prolog. The beta prologs consist of several symbols to the right of the sliding window. The alpha prolog consists of the several bits to the left of the sliding window. This is shown by the overlap between successive sliding blocks in the figure. Each beta sliding window is processed in reverse relative to the alpha sliding blocks.

Figure 9 shows an example of the order in which beta and alpha bits are processed. This example assumes a sliding window size of 100, a prolog length of 24, and 4 sliding windows per sub-block. The sliding block beta0 begins at the start of the prolog at bit 123. Next, the prolog ends at bit 100. The reliability data begins at bit 99, and ends at bit zero. The alpha sliding blocks are similarly divided. (Note the first two entries for alpha do not exist, because there is no prolog for the beginning of the block since the start and end points are known.)

The extrinsic cannot be processed in parallel with both the alpha and beta generation processes, because the extrinsic input data, which requires data from alpha, beta, and the parity data, must be input in a certain order. The following shows the indexing of the extrinsic input. E0 (corresponding to alpha0 and beta0) goes from bit 0 to 99. E1 goes from 100 to 199, and so on, given a sliding window size of 100. The input required by this example would be as follows. In the first clock cycle, the soft estimate data relating to bit 0 from alpha, beta, and P are input to the extrinsic. In the second clock cycle, data associated with bit 100 from the three inputs is required. In the third clock cycle, the data associated with bit 200 is required. In the fourth clock cycle, the data associated with bit 300 is required. In the fifth clock cycle, the input reverts back to the data associated with bit 1 (the first clock cycle input shifted one bit). In the next cycle, the bit 101 data, and so on. Thus the betas must be stored in RAM after they are generated, because they are generated in a different order than the alpha bits and parity bits, and are not required at generation as are the alpha bits and parity bits. When the corresponding alphas and betas have been generated, the extrinsic may be calculated.

Definitions:

Following are short definitions of the usual meanings of some of the technical terms which are used in the present application. (However, those of ordinary skill will recognize whether the context requires a different meaning.) Additional definitions can be found in the standard technical dictionaries and journals.

MAX*: MAX* is a maximum finding approximation for the natural log function, given by the following equation:

$$\ln[e^{A}+e^{B}] \approx \mathrm{MAX}^{*} = \mathrm{MAX}(A+B)+f(|A-B|)$$

where f(A-B) is a correction term. A lookup table is usually used for this value, which makes the above expression an approximation. If the expression

$$\ln[1+e^{-|A-B|}]$$

is used instead of a lookup table, then the MAX* definition becomes an exact equality, not an approximation.

MAP decoder: Maximum A-Posteriori. MAP decoders use a detection criterion that leads to the selection of x that maximizes the probability *p(x/r)* of a symbol x given the received information r.
Extrinsic: Outputs of decoders that estimate the value of a decoded bit.

Extrinsics are usually soft estimates.

Modifications and Variations

As will be recognized by those skilled in the art, the innovative concepts described in the present application can be modified and varied over a tremendous range of applications, and accordingly the scope of patented subject matter is not limited by any of the specific exemplary teachings given, but is only defined by the issued claims.

Though the preferred embodiment is given in specific detail, many alterations can be made in its implementation without escaping the scope of the inventive concepts herein disclosed. For instance, the latency of each state vector generation stage can be varied (by adding registers, or other means), and thus the degree of possible pipelining will vary. The size of the trellis can also be changed without altering the inventive concepts applied in the embodiment. The betas, alphas, and extrinsics may be generated in various parallel combinations, with only minor changes in RAM storage required.

Those of skill in the art will know that the definitions of inputs used in the present application (the systematic data X, and the parity data P) may be generalized to cover a broader range of applications. For instance, these inputs may differ in such applications as MAP equalization or turbo trellis decoding. In some applications, the inputs may not be soft estimates of bits, but rather they may be soft estimates of other variables. The disclosed innovations are intended to cover all such variations in implementation.

The disclosed innovations of the present application are applicable to any MAP architecture. For instance, any implementation of the disclosed inventive concepts in turbo decoders which use MAP decoders is within the contemplation of the invention. Any MAP operations, e.g., MAP equalization, are within the contemplation of the present application. MAP equalization is the process of describing the channel function as the data input to the channel constrained on a trellis to produce the observed output. The input to the channel can then be estimated in a maximum a priori sense by applying a MAP decode to the trellis diagram and the observed channel output. This is useful if (a) soft output is required from the equalizer, (b) a more accurate estimate of the input to the channel is required than can be got using a linear filter or equalizer, or (c) an iterative joint decode of the channel and the applied FEC is required. In general, MAP finds use in any situation where the data observed is known to have been generated by input to a linear trellis.

Likewise, MAP architectures with software, as well as hardware, implementations is within the contemplation of the invention. In today's DSPs very high processing rates are achieved by using deep pipelining of the data path. This means the DSP cannot be efficiently used in a feedback process such as beta and alpha updates. Using the present invention allows several blocks to be simultaneously processed by the DSP in a pipelined fashion, which considerably speeds up the operation in a deeply pipelined DSP architecture.

Further background material on the state of the art in MAP decoders and coding can be found in TURBO CODING, by Heegard and Wicker; TRELLIS CODING, by Schlegel; ERROR CONTROL SYSTEMS, by Wicker, and AN INTUITIVE JUSTIFICATION AND A SIMPLIFIED IMPLEMENTATION OF THE MAP DECODER FOR CONVOLUTIONAL CODES, Andrew Viterbi, IEEE Journal on Selected Areas of Communications, Vol. 16, No. 2, February 1998, all of which are hereby incorporated by reference.